import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem'; import UniV2FactoryABI from "@site/docs/specs/contracts/abi/_uniswapV2Factory.mdx"; import FlashLoanArbABI from "@site/docs/specs/contracts/abi/_flashLoanArb.mdx";

Automated Arbitrage Bot

With the flash-loan arbitrage smart contract finished, our last task is to automate the process of finding and back-running other users' trades.

Since we already have a full bot that you can look at for a complete example, we'll just lay out the core principles and methods here, without walking through a whole bot step by step. This guide aims to give you everything you need to write your own edition of simple-blind-arbitrage.

For a more in-depth guide on writing a bot from scratch, check out theMEV-Share Limit Order Bot tutorial.

Client Libraries

To write your own bot, you'll need a way to interact with Flashbots APIs. Client libraries implement all the functionality of the APIs in their native programming language. If your preferred language doesn't have a MEV-Share client library already, you can interact with the APIs directly, effectively implementing your own client (let us know if you do!). Information about Flashbots APIs can be found in the RPC docs and Event Stream docs.

Client libraries have been developed for the following programming languages (more to come soon):

||| |-|-| | typescript/javascript |mev-share-client-ts (npm) | | rust |mev-share-rs (crates.io) |

Finding Pending Transactions

To reiterate our goal, we need to find pending transactions from other Ethereum users and backrun them (send our transaction immediately after the user's transaction). If the user's trade moves the price enough, we'll arbitrage the trading pair between two exchanges for a profit.

As MEV-Share receives new transactions, it broadcasts them to searchers via theEvent Stream. Each transaction shares varying amounts of data via hints, but by default, transactions that trade on Uniswap, Balancer, or Curve will expose the trading pair's contract address in the log topics.

Here's an example of an event generated by a user who's (most likely) using MEV-Share's default privacy settings:

Example Event

json noInline { "hash": "0x0c459dce812747c643f06c82eeca2a2f584b4e30af79b2b546fd015e4aac4541", "logs": [{ "address": "0xca25091555d36ac0be8119ad967898ac30223b41", "topics": ["0xd78ad95fa46c994b6551d0da85fc275fe613ce37657fb8d5e3d130840159d822", "0x0000000000000000000000000000000000000000000000000000000000000000", "0x0000000000000000000000000000000000000000000000000000000000000000"], "data": "0x" }, { "address": "0xca25091555d36ac0be8119ad967898ac30223b41", "topics": ["0xd78ad95fa46c994b6551d0da85fc275fe613ce37657fb8d5e3d130840159d822", "0x0000000000000000000000000000000000000000000000000000000000000000", "0x0000000000000000000000000000000000000000000000000000000000000000"], "data": "0x" }], "txs": null }

If we look up the address specified in the log topics, we'll see that it's a trading pair contract for QPEPE/WETH on Uniswap V2. This tells us that the user is buying/selling QPEPE on Uniswap, which means that we may be able to arbitrage it with Sushiswap or another Uni-V2 derivative.

To detect events yourself, listen to the SSE Event Stream athttps://mev-share.flashbots.net.

Events are comprised of either a single**transaction** or a **bundle** (multiple transactions):

- In a **bundle** event, the `hash` field is the *bundle hash* and the `txs` field will be populated with transaction-related data for each tx in the bundle.
- In a single-**transaction** event, `hash` represents the *transaction hash* and the `txs` field is `null`.

```typescript const authSigner = new Wallet(Env.authKey).connect(provider) const mevshare = MevShareClient.useEthereumMainnet(authSigner)

mevshare.on("transaction", async (pendingTx: IPendingTransaction) => { // handle tx event }) ```

In this library, transactions are natively differentiated from bundles, so if you want to detect bundles too, just add another handler using `on("bundle", ...)`:

typescript mevshare.on("bundle", async (pendingBundle: IPendingBundle) => { // handle bundle event })
```

Check out the implementation for a closer look at how events are defined.

rust let mainnet_sse = "https://mev-share.flashbots.net"; let client = EventClient::default(); let mut stream = client.events(mainnet_sse).await.unwrap(); while let Some(event) = stream.next().await { // handle event }

In this library, an event whose `txs` field is null is encoded as an empty array, so in practice, we need to check if `event.txs` is empty to see if we have a transaction or a bundle.

rust while let Some(event) = stream.next().await { if event.txs.len() == 0 { // handle single tx } else { // handle bundle } }

bash curl https://mev-share.flashbots.net

The Event Stream is streamed via SSE over a simple HTTP GET request, which may seem confusing. It's not terribly important for the purpose of building bots, but if you're curious, SSE was added to HTML in HTML5.

To read more about how the SSE stream works, see the Event Stream docs.

Filtering Relevant Transactions

MEV-Share uses **hints** to selectively share information about a transaction. Based on the hint preferences specifies by the user when connecting, the transactions (or bundles) they send will trigger events containing information about their transaction which is filtered according to their hint preferences.

In this guide, we're only concerned with the fields in `logs`: `address` and `topics`. Other fields not covered in this guide are detailed in the Event Scheme docs.

Snippet from an example event:

json { ..., "logs": [{ "address": "0xca25091555d36ac0be8119ad967898ac30223b41", "topics": ["0xd78ad95fa46c994b6551d0da85fc275fe613ce37657fb8d5e3d130840159d822", "0x0000000000000000000000000000000000000000000000000000000000000000", "0x0000000000000000000000000000000000000000000000000000000000000000"], "data": "0x" }, ...], }

address tells us which contract address the user is interacting with. Looking back at the example event, you'll notice that the `address` is the trading pair, and not a router contract. This is likely because the user is using the default hints, which expose the trading pair instead of whatever router they might be interacting with. It's also possible that the user is another searcher, and they're trading on the pair directly. At any rate, it makes no difference to us.

Given the default hint preferences, MEV-Share also exposes *only* swap-related function signatures (if present) in the `topics`. The simple-blind-arbitrage bot uses the Uniswap V2 Swap event signature to find pending Uniswap V2 trades.

To find event signatures yourself, you need to take the keccak256 hash of the event signature. A nice tool to use for this is **`cast`** from Foundry:

```bash
# get the hash of the UniV2 Swap event

cast sig-event "event Swap( address indexed sender, uint amount0In, uint amount1In, uint amount0Out, uint amount1Out, address indexed to );" 0xd78ad95fa46c994b6551d0da85fc275fe613ce37657fb8d5e3d130840159d822 ```

To filter our results so that we only deal with Swap events, we can simply check the `logs` field. If it isn't empty, each entry should contain `topics`. If our event signature hash is in those topics, then we know we're looking at a swap that we can backrun. For events generated by Protect users, the relevant signature hash is the first topic in the array, so we check `log.topics[0]`.

Once we find a pending swap on one exchange, we need to find another exchange to arbitrage with. For example, if we detect a Uniswap V2 event, then we should try to arbitrage with Sushiswap. To get the trading pair on the other exchange, we need to use that exchange's Factory contract, which maps pairs of token addresses to Pair contract addresses.

```typescript import {Contract} from "ethers" import uniV2FactoryABI from "./abi/uniswapV2Factory.json"

const swapTopic = "0xd78ad95fa46c994b6551d0da85fc275fe613ce37657fb8d5e3d130840159d822"

// instantiate factory contracts const uniV2FactoryAddress = "0x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f" const uniV2Factory = new Contract(uniV2FactoryAddress, uniV2FactoryABI) const sushiFactoryAddress = "0xC0AEe478e3658e2610c5F7A4A2E1777cE9e4f2Ac" const sushiFactory = new Contract(sushiFactoryAddress, uniV2FactoryABI)

mevshare.on("transaction", async (tx: IPendingTransaction) => { for (const log of tx.logs) { // skip if it isn't a swap event if

(log.topics[0] !== swapTopic) { continue }

// data needed to find arb pair on another exchange
const pair = new ethers.Contract(log.address, pairABI)
const token0 = await pair.token0()
const token1 = await pair.token1()
// primitive differentiator between uniswap & sushiswap. ideally you'd use an enum to support >2 exchanges.
const isUniswap = pair.factory === uniV2FactoryAddress

const altFactory = isUniswap ? sushiFactory : uniV2Factory
const altPair = await altFactory.getPair(token0, token1)
if (altPair === "0x0000000000000000000000000000000000000000") {
  console.error("pair not found on alternative exchange")
  continue
}

// placeholder: send backrun bundle
await tryBackrun(pair.address, altPair)

} }) ```

Note: querying the blockchain can cost precious time -- a better design would be to store arb-ready pairs somewhere fast, like in memory or a fast DB, and only query the blockchain when a stored pair alternative can't be found.

```rust let swap_topic = "0xd78ad95fa46c994b6551d0da85fc275fe613ce37657fb8d5e3d130840159d822" .parse::() .unwrap(); while let Some(event) = stream.next().await { dbg!(&event); // handle tx or bundle events containing swap logs let event = event.unwrap(); for log in event.logs { // skip if it isn't a swap event if log.topics[0] != swap_topic { continue; }

// get data needed to find arb pair
// TODO

// placeholder: send backrun bundle
tryBackrun(pair.address, altPair.address).await?;

} } ```

Note that the Sushiswap factory uses the same ABI as the Uniswap contract. Sushiswap is just a Uniswap V2 clone, so we know that the code is the same, which means the ABIs are also the same.

:::info blind backruns

Note that we potentially send a bundle for every log in the array. This is because some trades affect many trading pairs. Because we don't know which one was affected, we just attempt to backrun them all. The bundles we'll send for a given opportunity will use the same nonce for every backrun tx, so only the most profitable one will land.

:::

# Sending Backrun Bundles

So far we've seen how to find the events we want to backrun, and we have a placeholder `t(yBackrun)` in place to send our backrun bundles. Once we implement `tryBackrun`, we'll have everything we need to send backrun bundles.

A backrun bundle is an array consisting of two or more transactions: the user's transaction, and the following "backrun transactions." The backrun transaction we'll send is a call to `makeFlashLoan` on our arbitrage contract. If you recall from the implementation of the flash loan arbitrage contract, the `receiveFlashLoan` function calls `_executeArbitrage` before paying back the loan. So all we have to do is get a flash loan, and then our contract will try to execute an arbitrage with the tokens it receives. If it's profitable, we pay the loan back, pay the builder, and keep the rest. If not, it reverts and is discarded by the builder.

*This is the function we need to call:*

solidity function makeFlashLoan( IERC20[] memory tokens, uint256[] memory amounts, bytes memory userData )

To call `makeFlashLoan`, we specify the `tokens` we want to borrow, the respective `amounts` of tokens to borrow, and the ABI-encoded `userData` which will contain the arguments to the `_executeArbitrage` function.

This function sends a backrun given two pair addresses and a pending tx hash from the Event Stream:

```typescript import ArbContractAbi from ".abi/blindBackrunFlashLoan.json"

const arbContractAddress = "0xTODO" const arbContract = new Contract(arbContractAddress, ArbContractAbi, provider)

async function tryBackrun(startPair, endPair, txHash) { let blockNumber = Number(await this.signer.provider.getBlockNumber()) console.log("Current block number:", blockNumber) console.log("Building bundles")

let bundleTransactionOptions = { gasPrice: (await this.signer.provider.getGasPrice()), // This is *extremely* naive. gasLimit: ethers.BigNumber.from(400000), nonce: await this.signer.getTransactionCount(), } const types = [ 'address', 'address', 'uint256' ] const values = [ startPair, endPair, this.percentageToKeep ] let params = Web3EthAbi.encodeParameters(types, values)

let bundleTransaction = await arbContract.populateTransaction.makeFlashLoan( config.mainnetWETHAddress, ethers.BigNumber.from(10**21).toString(), params, bundleTransactionOptions ) let bundle = [ {hash: txHash}, {tx: await this.signer.signTransaction(bundleTransaction), canRevert: false}, ] let fullBundle = { inclusion: { block: blockNumber + 1, // try to land in next block maxBlock: ethers.utils.hexValue(blockNumber + 24) // Protect txs expire after 25 blocks }, body: bundle, }

return await mevshare.sendBundle(fullBundle)

// for the reader: also send another bundle that calls makeFlashLoan // with startPair and endPair in switched in values. } ```

We get blockNumber at the start to set the target block in our bundle params bundleTransactionOptions is pretty self-explanatory -- we use this because we're signing our backrun transaction manually, so it needs these params to be set manually.

types and values define the function params that are passed via userData (as the variable params) in makeFlashLoan, and then decoded in receiveFlashLoan and passed to _executeArbitrage.

We use arbContract.populateTransaction.makeFlashLoan to construct the backrun transaction, and then build a bundle with it and the user's pending tx hash. Lastly, we add the full bundle params for the MEV-Share sendBundle call, and then call it and return its result.

mev-share-rs

typescript // rs

// TODO: Explain specific rust code here.

Put these examples together and you should have a working flash-loan arbitrage bot. If you get stuck, checkout simple-blind-arbitrage, written in javascript.